Data Structures with Python

Azure

2022

Contents

1 Linked Lists

1.1 Singly Linked List

```python
class node:
    def __init__(self, data = None, next = None):
        self.data = data
        self.next = next

class linkedlist:
    def __init__(self):
        self.head = None
        self.tail = self.head
        self.length = 0

    def push_front(self, data):
        if (self.head is None):
            self.head = node(data, None)
            self.tail = self.head
        else:
            self.head = node(data, self.head)
        self.length += 1

    def find_nth(self, index):
        if (index >= self.length):
            return None
        else:
            cur_index = 0
            cur = self.head
            while (cur_index < index):
                cur = cur.next
                cur_index += 1
            return cur

    def get_nth(self, index):
        if (index >= self.length):
            return None
        else:
            return self.find_nth(index).data

    def set_nth(self, index, data):
        if (index >= self.length):
            return None
        else:
            self.find_nth(index).data = data
```

1.2 Doubly Linked List

```python
class node:
    def __init__(self, data = None, prev = None, next = None):
        self.data = data
        self.prev = prev
        self.next = next

class linkedlist:
    def __init__(self):
        self.head = None
        self.tail = self.head
        self.length = 0

    def push_front(self, data):
        if (self.head is None):
            self.head = node(data, None, None)
            self.tail = self.head
        else:
            self.head.next = node(data, self.head, None)
            self.head = self.head.next
        self.length += 1

    def push_back(self, data):
        if (self.head is None):
            self.head = node(data, None, None)
            self.tail = self.head
        else:
            self.tail.prev = node(data, None, self.tail)
            self.tail = self.tail.prev
        self.__length += 1

    def pop_front(self):
        if (self.__length == 0):
            return None
        elif (self.__length == 1):
            value = self.head.data
            self.head = None
            self.tail = None
        else:
            value = self.head.data
            self.head = self.head.prev
            self.head.next = None
        self.length -= 1
        return value

    def pop_back(self):
        if (self.length == 0):
            return None
        elif (self.length == 1):
            value = self.tail.data
            self.head = None
            self.tail = None
        else:
            value = self.tail.data
            self.tail = self.tail.next
            self.head.prev = None
        self.length -= 1
        return value
```

2 Searches & Sorting

2.1 Linear Search

```
def linear_search(list, value):
    for i in range(0, len(list), 1):
        if (list[i] == value):
            return i
    return None
```

2.2 Binary Search

```
def binary_search(list, value):
    def limits(min, max):
        mid = (min + max) // 2
        if (min > max):
            return None
        elif (list[mid] < value):
            return limits(mid+1, max)
        elif (list[mid] > value):
            return limits(min, mid-1)
        else:
            return mid
    return limits(0, len(list)-1)
```

2.3 Selection Sort

```
def selection_sort2(list):
    temp = list.copy()
    for i in range(0, len(temp), 1):
        min = temp[i]
        for j in range(i+1, len(temp), 1):
            if (temp[j] < temp[i]):
                min = temp[j]
                temp[j] = temp[i]
                temp[i] = min
    return list
```

2.4 Merge Sort

```
def sort(list1, list2):
    temp = list1 + list2
    i = 0
    j = len(list1)
    list_sorted = []
    while (i != len(list1) and j != len(temp)):
        if (temp[i] <= temp[j]):
            list_sorted.append(temp[i])
            i += 1
        else:
            list_sorted.append(temp[j])
            j += 1
    if (i != len(list1)):
        list_sorted = list_sorted + temp[i:len(list1)]
    else:
        list_sorted = list_sorted + temp[j:len(temp)]
    return list_sorted

def merge_sort(list):
    if (len(list) <= 1):
        return list
    else:
        mid = len(list) // 2
        lista = list[0:mid]
        listb = list[mid:len(list)]
        return sort(merge_sort(lista), merge_sort(listb))
```

3 Stacks & Queues

3.1 Stack

```
class node:
    def __init__(self, data = None, next = None):
        self.data = data
        self.next = next

class stack:
    def __init__(self):
        self.head = None
        self.length = 0

    def push(self, data):
        self.head = node(data, self.head)
        self.length += 1

    def pop(self):
        if (self.length == 0):
            return None
        else:
            value = self.head.data
            self.head = self.head.next
            self.length -= 1
            return value

    def empty(self):
        return self.length == 0
```

3.2 Queue

```
class queue:
    def __init__(self):
        self.head = None
        self.tail = self.head
        self.length = 0

    def enqueue(self, data):
        if (self.head is None):
            self.head = node(data, self.head)
            self.tail = self.head
        else:
            self.tail.next = node(data, None)
            self.tail = self.tail.next
        self.length += 1

    def dequeue(self):
        if (self.length == 0):
            return None
        else:
            value = self.head.data
            self.head = self.head.next
            self.length -= 1
            return value

    def empty(self):
        return self.length == 0
```

4 Dictionaries

```
class entry:
    def __init__(self, key, value, next = None):
        self.key = key
        self.value = value
        self.next = next

# Dictionary (key-value pairs) using a hash function
class dict:
    def __init__(self):
        self.size = 32
        self.length = 0
        self.scale = self.length / self.size
        self.data = [None] * self.size

    def add_entry(self, key, value):
        if (self.get_value(key) is None):
            self.length += 1
            self.scale = self.length / self.size
            self.check_grow()
            pos = hash(key) % self.size
            self.data[pos] = entry(key, value, self.data[pos])
        else:
            pos = hash(key) % self.size
            bucket = self.data[pos]
            while (bucket is not None):
                if (bucket.key == key):
                    bucket.value = value
                else:
                    bucket = bucket.next

    def get_value(self, key):
        pos = hash(key) % self.size
        bucket = self.data[pos]
        while (bucket is not None):
            if (bucket.key == key):
                return bucket.value
            else:
                bucket = bucket.next
        return None

    def check_grow(self):
        if (self.scale > 2):
            self.size = self.size * 2
            new_data = [None] * self.size
            for bucket in self.data:
                cur_bucket = bucket
                while (cur_bucket is not None):
                    pos = hash(cur_bucket.key) % self.size
                    new_data[pos] = entry(cur_bucket.key,
                                          cur_bucket.value,
                                          new_data[pos])
                    cur_bucket = cur_bucket.next
            self.data = new_data
            self.scale = self.length / self.size
```

5 Graphs

5.1 Graph

```
class graph:
    def __init__(self):
        self.vertices = []
        self.edges = []
        self.length = 0

    def add_vertice(self, vertice):
        if (vertice not in self.vertices):
            self.length += 1
            self.vertices.append(vertice)
            for x in self.edges:
                x.append(False)
            self.edges.append([False] * self.length)
            self.edges[self.length - 1][self.length - 1] = True

    def set_connect(self, vertice1, vertice2, value):
        if (vertice1 in self.vertices and vertice2 in self.vertices):
            pos1 = self.vertices.index(vertice1)
            pos2 = self.vertices.index(vertice2)
            self.edges[pos1][pos2] = value
            self.edges[pos2][pos1] = value

    def connect(self, vertice1, vertice2):
        self.set_connect(vertice1, vertice2, True)

    def disconnect(self, vertice1, vertice2):
        self.set_connect(vertice1, vertice2, False)

    def has_edge(self, vertice1, vertice2):
        if (vertice1 in self.vertices and vertice2 in self.vertices):
            pos1 = self.vertices.index(vertice1)
            pos2 = self.vertices.index(vertice2)
            return self.edges[pos1][pos2]
        else:
            return False

    def get_neighbors(self, vertice):
        temp = []
        if (vertice in self.vertices):
            pos = self.vertices.index(vertice)
            for i in range(0, self.length):
                if (i != pos and self.edges[pos][i]):
                    temp.append(self.vertices[i])
        return temp
```

5.2 Directed Graph

```
class digraph:
    def __init__(self):
        self.vertices = []
        self.edges = []
        self.length = 0

    def add_vertice(self, vertice):
        if (vertice not in self.vertices):
            self.length += 1
            self.vertices.append(vertice)
            for x in self.edges:
                x.append(False)
            self.edges.append([False] * self.length)
            self.edges[self.length - 1][self.length - 1] = True

    def set_connect(self, vertice1, vertice2, value):
        if (vertice1 in self.vertices and vertice2 in self.vertices):
            pos1 = self.vertices.index(vertice1)
            pos2 = self.vertices.index(vertice2)
            self.edges[pos1][pos2] = value

    def connect(self, vertice1, vertice2):
        self.set_connect(vertice1, vertice2, True)

    def disconnect(self, vertice1, vertice2):
        self.set_connect(vertice1, vertice2, False)

    def has_edge(self, vertice1, vertice2):
        if (vertice1 in self.vertices and vertice2 in self.vertices):
            pos1 = self.vertices.index(vertice1)
            pos2 = self.vertices.index(vertice2)
            return self.edges[pos1][pos2]
        else:
            return False

    def get_neighbors(self, vertice):
        temp = []
        if (vertice in self.vertices):
            pos = self.vertices.index(vertice)
            for i in range(0, self.length):
                if (i != pos and self.edges[pos][i]):
                    temp.append(self.vertices[i])
        return temp
```

5.3 Weighted Directed Graph

```
class wdigraph:
    def __init__(self):
        self.vertices = []
        self.edges = []
        self.length = 0

    def add_vertice(self, vertice):
        if (vertice not in self.vertices):
            self.length += 1
            self.vertices.append(vertice)
            for x in self.edges:
                x.append(float("inf"))
            self.edges.append([float("inf")] * self.length)
            self.edges[self.length - 1][self.length - 1] = 0

    def set_connect(self, vertice1, vertice2, value):
        if (vertice1 in self.vertices and vertice2 in self.vertices):
            pos1 = self.vertices.index(vertice1)
            pos2 = self.vertices.index(vertice2)
            self.edges[pos1][pos2] = value

    def connect(self, vertice1, vertice2, value):
        self.set_connect(vertice1, vertice2, value)

    def disconnect(self, vertice1, vertice2):
        self.set_connect(vertice1, vertice2, float("inf"))

    def get_edge(self, vertice1, vertice2):
        if (vertice1 in self.vertices and vertice2 in self.vertices):
            pos1 = self.vertices.index(vertice1)
            pos2 = self.vertices.index(vertice2)
            return self.edges[pos1][pos2]
        else:
            return float("inf")

    def has_edge(self, vertice1, vertice2):
        if (self.get_edge(vertice1, vertice2) == float("inf")):
            return False
        else:
            return True

    def get_neighbors(self, vertice):
        temp = []
        if (vertice in self.vertices):
            pos = self.vertices.index(vertice)
            for i in range(0, self.length):
                if (i != pos and self.edges[pos][i] != float("inf")):
                    temp.append(self.vertices[i])
        return temp
```

6 Graph Searches

6.1 Depth-First Search

```
def dfs(graph, start):
    visited = [None] * graph.length
    to_visit_stack = stack()
    visited_from_stack = stack()

    to_visit_stack.push(start)
    visited_from_stack.push(start)

    while (not to_visit_stack.empty()):
        from_vertex = to_visit_stack.pop()
        vertex_index = graph.get_index(from_vertex)
        if (visited[vertex_index] is None):
            visited[vertex_index] = visited_from_stack.pop()
            next_vertices = graph.get_neighbors(from_vertex)
            for v in next_vertices:
                to_visit_stack.push(v)
                visited_from_stack.push(from_vertex)
        else:
            visited_from_stack.pop()
    return visited
```

6.2 Breath-First Search

```
def bfs(graph, start):
    visited = [None] * graph.length
    to_visit_queue = queue()
    visited_from_queue = queue()

    to_visit_queue.enqueue(start)
    visited_from_queue.enqueue(start)

    while (not to_visit_queue.empty()):
        from_vertex = to_visit_queue.dequeue()
        vertex_index = graph.get_index(from_vertex)
        if (visited[vertex_index] is None):
            visited[vertex_index] = visited_from_queue.dequeue()
            next_vertices = graph.get_neighbors(from_vertex)
            for v in next_vertices:
                to_visit_queue.enqueue(v)
                visited_from_queue.enqueue(from_vertex)
        else:
            visited_from_queue.dequeue()
    return visited
```

7 Single Source Shortest Path

7.1 Depth-First Search Method

```
def dfs_sssp(graph, start):
    edges = graph.get_edges()
    for vertice_pair in edges:
        for weight in vertice_pair:
            if (weight < 0):
                return False

    distance = [float("inf")] * graph.get_length()
    visited = [None] * graph.get_length()
    to_visit_stack = stack()

    start_index = graph.get_index(start)
    distance[start_index] = 0
    visited[start_index] = start

    to_visit_stack.push(start)

    while (not to_visit_stack.empty()):
        cur_vertex = to_visit_stack.pop()
        cur_index = graph.get_index(cur_vertex)

        next_vertices = graph.get_neighbors(cur_vertex)
        for v in next_vertices:
            cur_distance = distance[cur_index] + graph.get_edge(cur_vertex, v)
            next_index = graph.get_index(v)
            if (cur_distance < distance[next_index]):
                distance[next_index] = cur_distance
                visited[next_index] = cur_vertex
                to_visit_stack.push(v)
    return visited
```

7.2 Djikstra's Algorithm

```python
class wnode:
    def __init__(self, data = None, value = None, next = None):
        self.data = data
        self.value = value
        self.next = next

# Priority Queue
class priority_queue:
    def __init__(self):
        self.head = None
        self.tail = self.head
        self.length = 0

    def enqueue(self, data, priority):
        if (self.head is None):
            self.head = wnode(data, priority, self.head)
            self.tail = self.head
        else:
            if (self.tail.value < priority):
                self.tail.next = wnode(data, priority, None)
                self.tail = self.tail.next
            elif (self.head.value >= priority):
                self.head = wnode(data, priority, self.head)
            else:
                curr_data = self.head
                while (curr_data.next.value < priority):
                    curr_data = curr_data.next
                curr_data.next = wnode(data, priority, curr_data.next)
        self.length += 1

    def dequeue(self):
        if (self.length == 0):
            return None
        else:
            value = self.head.data
            self.head = self.head.next
            self.length -= 1
            return value

    def empty(self):
        return self.length == 0
```

```python
def dijkstra(graph, start):
    edges = graph.get_edges()
    for vertice_pair in edges:
        for weight in vertice_pair:
            if (weight < 0):
                return False

    distance = [float("inf")] * graph.get_length()
    visited = [None] * graph.get_length()
    to_visit = priority_queue()
    completed = [False] * graph.get_length()

    start_index = graph.get_index(start)
    distance[start_index] = 0
    visited[start_index] = start

    to_visit.enqueue(start, 0)

    while (not to_visit.empty()):
        cur_vertex = to_visit.dequeue()
        cur_index = graph.get_index(cur_vertex)

        if (not completed[cur_index]):
            completed[cur_index] = True
            next_vertices = graph.get_neighbors(cur_vertex)
            for v in next_vertices:
                cur_dist = distance[cur_index] + graph.get_edge(cur_vertex, v)
                next_index = graph.get_index(v)
                if (cur_dist < distance[next_index]):
                    distance[next_index] = cur_dist
                    visited[next_index] = cur_vertex
                    priority = graph.get_edge(cur_vertex, v)
                    to_visit.enqueue(v, priority)
    return visited
```

7.3 Binary Heap

```
class bin_heap:
    def __init__(self):
        self.size = 32
        self.length = 0
        self.keys = [None] * self.size
        self.values = [float("inf")] * self.size

    def insert(self, key, value: int):
        if (self.length == 0):
            self.keys[0] = key
            self.values[0] = value
        else:
            cur_index = self.length
            parent_index = (cur_index - 1) // 2
            self.keys[cur_index] = key
            self.values[cur_index] = value
            while (parent_index >= 0 and value < self.values[parent_index]):
                self.values[cur_index] = self.values[parent_index]
                self.keys[cur_index] = self.keys[parent_index]
                self.values[parent_index] = value
                self.keys[parent_index] = key
                cur_index = parent_index
                parent_index = (cur_index - 1) // 2
        self.length += 1
        self.check_grow()

    def remove(self):
        if (self.length == 0):
            return None
        else:
            key = self.keys[0]
            cur_value = self.values[self.length - 1]
            cur_key = self.keys[self.length - 1]
            self.values[self.length - 1] = float("inf")
            self.keys[self.length - 1] = float("inf")
            cur_index = 0
            self.values[cur_index] = cur_value
            self.keys[cur_index] = cur_key
            left = 2 * cur_index + 1
            right = 2 * cur_index + 2
            while (left < self.size):
                if (cur_value > self.values[left] and
                    cur_value > self.values[right]):
                    if (self.values[left] <= self.values[right]):
                        self.values[cur_index] = self.values[left]
                        self.keys[cur_index] = self.keys[left]
                        self.values[left] = cur_value
                        self.keys[left] = cur_key
                        cur_index = left
                    else:
                        self.values[cur_index] = self.values[right]
                        self.keys[cur_index] = self.keys[right]
                        self.values[right] = cur_value
                        self.keys[right] = cur_key
                        cur_index = right
                elif (cur_value > self.values[left]):
                    self.values[cur_index] = self.values[left]
                    self.keys[cur_index] = self.keys[left]
                    self.values[left] = cur_value
                    self.keys[left] = cur_key
                    cur_index = left
                elif (cur_value > self.values[right]):
                    self.values[cur_index] = self.values[right]
                    self.keys[cur_index] = self.keys[right]
                    self.values[right] = cur_value
                    self.keys[right] = cur_key
                    cur_index = right
                else:
                    break
                left = 2 * cur_index + 1
                right = 2 * cur_index + 2
            self.length -= 1
            return key

    def check_grow(self):
        if (self.length == self.size):
            self.keys = self.keys + [None] * self.size
            self.values = self.values + [float("inf")] * self.size
            self.size *= 2

    def get_keys(self):
        return self.keys

    def get_values(self):
        return self.values

    def empty(self):
        return (self.length == 0)
```

```
def dijkstra_heap(graph, start):
    edges = graph.get_edges()
    for vertice_pair in edges:
        for weight in vertice_pair:
            if (weight < 0):
                return False

    distance = [float("inf")] * graph.get_length()
    visited = [None] * graph.get_length()
    to_visit = bin_heap()
    completed = [False] * graph.get_length()

    start_index = graph.get_index(start)
    distance[start_index] = 0
    visited[start_index] = start

    to_visit.insert(start, 0)

    while (not to_visit.empty()):
        cur_vertex = to_visit.remove()
        cur_index = graph.get_index(cur_vertex)

        if (not completed[cur_index]):
            completed[cur_index] = True
            next_vertices = graph.get_neighbors(cur_vertex)
            for v in next_vertices:
                cur_dist = distance[cur_index] + graph.get_edge(cur_vertex, v)
                next_index = graph.get_index(v)
                if (cur_dist < distance[next_index]):
                    distance[next_index] = cur_dist
                    visited[next_index] = cur_vertex
                    priority = graph.get_edge(cur_vertex, v)
                    to_visit.insert(v, priority)
    return visited
```

8 Minimum Spanning Trees

8.1 Union-Find: Quick-Find Version

```
class union_find_qf:
    def __init__(self, length):
        self.elements = list(range(0, length))
        self.length = length

    def get_parent(self, value):
        return self.elements[value]

    def is_same_set(self, value1, value2):
        return self.get_parent(value1) == self.get_parent(value2)

    def union(self, value1, value2):
        convert_from = self.get_parent(value1)
        convert_to = self.get_parent(value2)
        if (convert_from != convert_to):
            for value in range(0, self.length):
                if (self.get_parent(value) == convert_from):
                    self.elements[value] = convert_to

    def get_set(self):
        return self.elements

def kruskal(graph):
    min_graph = wgraph()
    graph_vertices = graph.get_vertices()
    for vertex in graph_vertices:
        min_graph.add_vertice(vertex)
    components = union_find_qf(graph.get_length())
    edges_to_check = bin_heap()
    graph_edges = graph.get_edges()
    for vertex_from_index in range(0, graph.get_length()):
        for vertex_to_index in range(0, graph.get_length()):
            if (graph_edges[vertex_from_index][vertex_to_index] < float("inf")
                and vertex_from_index != vertex_to_index):
                edges_to_check.insert(
                    [vertex_from_index, vertex_to_index],
                    graph_edges[vertex_from_index][vertex_to_index])

    while (not edges_to_check.empty()):
        cur_edge = edges_to_check.remove()
        if (not components.is_same_set(cur_edge[0], cur_edge[1])):
            components.union(cur_edge[0], cur_edge[1])
            min_graph.connect(
                graph_vertices[cur_edge[0]],
                graph_vertices[cur_edge[1]],
                graph_edges[cur_edge[0]][cur_edge[1]])

    return min_graph
```

8.2 Union-FInd: Weighted Quick-Union Version

```
class union_find_wqu:
    def __init__(self, length: int):
        self.elements = list(range(0, length))
        self.weights = [1] * length
        self.length = length

    def get_parent(self, value):
        cur_parent = self.elements[value]
        while (self.elements[cur_parent] != cur_parent):
            cur_parent = self.elements[cur_parent]
        return cur_parent

    def is_same_set(self, value1, value2):
        return self.get_parent(value1) == self.get_parent(value2)

    def union(self, value1, value2):
        root1 = self.get_parent(value1)
        root2 = self.get_parent(value2)
        if (root1 != root2):
            if (self.weights[root1] <= self.weights[root2]):
                self.elements[root1] = root2
                self.weights[root2] += self.weights[root1]
            else:
                self.elements[root2] = root1
                self.weights[root1] += self.weights[root2]

    def get_set(self):
        return self.elements

    def get_weights(self):
        return self.weights
```

8.3 Union-Find: Weighted Quick-Union + Path Compression

```
class union_find_wqupc:
    def __init__(self, length: int):
        self.elements = list(range(0, length))
        self.weights = [1] * length
        self.length = length

    def get_parent(self, value):
        prev_value = value
        cur_value = self.elements[prev_value]
        elements_passed_through = []
        cur_weight_total = 0
        while (self.elements[cur_value] != cur_value):
            elements_passed_through.append(prev_value)
            cur_weight_total += self.weights[prev_value]
            self.weights[cur_value] -= cur_weight_total
            prev_value = cur_value
            cur_value = self.elements[cur_value]
        for element in elements_passed_through:
            self.elements[element] = cur_value
        return cur_value

    def is_same_set(self, value1, value2):
        return self.get_parent(value1) == self.get_parent(value2)

    def union(self, value1, value2):
        root1 = self.get_parent(value1)
        root2 = self.get_parent(value2)
        if (root1 != root2):
            if (self.weights[root1] <= self.weights[root2]):
                self.elements[root1] = root2
                self.weights[root2] += self.weights[root1]
            else:
                self.elements[root2] = root1
                self.weights[root1] += self.weights[root2]

    def get_set(self):
        return self.elements

    def get_weights(self):
        return self.weights
```

9 Binary Search Trees

9.1 Binary Search Tree

```
class node:
    def __init__(self, value, left = None, right = None):
        self.value = value
        self.left = left
        self.right = right

class bst:
    def __init__(self):
        self.data = None

    def insert(self, value: int):
        if (self.data is None):
            self.data = node(value, None, None)
        else:
            cur_node = self.data
            while (cur_node.value != value):
                if (value < cur_node.value):
                    if (cur_node.left is not None):
                        cur_node = cur_node.left
                    else:
                        break
                else:
                    if (cur_node.right is not None):
                        cur_node = cur_node.right
                    else:
                        break
            if (value < cur_node.value):
                cur_node.left = node(value, None, None)
            elif (value > cur_node.value):
                cur_node.right = node(value, None, None)

    def search(self, value: int):
        cur_node = self.data
        while (cur_node is not None):
            if (value < cur_node.value):
                cur_node = cur_node.left
            elif (value > cur_node.value):
                cur_node = cur_node.right
            else:
                return True
        return False

    def delete(self, value: int):
        prev_node = self.data
        cur_node = prev_node
        while (cur_node is not None):
            if (value < cur_node.value):
                prev_node = cur_node
                cur_node = cur_node.left
            elif (value > cur_node.value):
                prev_node = cur_node
                cur_node = cur_node.right
            else:
                break
        if (cur_node is not None):
            if (prev_node.value == cur_node.value):
```

```
            if (cur_node.right is None and cur_node.left is None):
                self.data = None
            elif (cur_node.right is not None):
                new_prev_node = cur_node
                cur_node = cur_node.right
                if (cur_node.left is None):
                    new_prev_node.value = cur_node.value
                    new_prev_node.right = cur_node.right
                else:
                    while (cur_node.left is not None):
                        new_prev_node = cur_node
                        cur_node = cur_node.left
                    prev_node.value = cur_node.value
                    new_prev_node.left = None
            else:
                new_prev_node = cur_node
                cur_node = cur_node.left
                if (cur_node.right is None):
                    new_prev_node.value = cur_node.value
                    new_prev_node.left = cur_node.left
                else:
                    while (cur_node.right is not None):
                        new_prev_node = cur_node
                        cur_node = cur_node.right
                    prev_node.value = cur_node.value
                    new_prev_node.right = None
        elif (cur_node.left is None and cur_node.right is None):
            if (prev_node.left.value == value):
                prev_node.left = None
            else:
                prev_node.right = None
        elif (cur_node.right is not None):
            new_prev_node = cur_node
            cur_node = cur_node.right
            if (cur_node.left is None):
                new_prev_node.value = cur_node.value
                new_prev_node.right = cur_node.right
            else:
                while (cur_node.left is not None):
                    new_prev_node = cur_node
                    cur_node = cur_node.left
                if (prev_node.left.value == value):
                    prev_node.left.value = cur_node.value
                else:
                    prev_node.right.value = cur_node.value
                new_prev_node.left = None
        else:
            new_prev_node = cur_node
            cur_node = cur_node.left
            if (cur_node.right is None):
                new_prev_node.value = cur_node.value
                new_prev_node.left = cur_node.left
            else:
                while (cur_node.right is not None):
                    new_prev_node = cur_node
                    cur_node = cur_node.right
                if (prev_node.left.value == value):
                    prev_node.left.value = cur_node.value
                else:
                    prev_node.right.value = cur_node.value
                new_prev_node.right = None

def get_data(self):
    return self.data
```

9.2 AVL Trees: Self-Balancing Binary Tree

```
class node:
    def __init__(self, value = None, height = 1,
                up_left = None, up_right = None,
                down_left = None, down_right = None):
        self.value = value
        self.height = height
        self.up_left = up_left
        self.up_right = up_right
        self.down_left = down_left
        self.down_right = down_right

    def get_balance(self):
        if (self.down_left is not None and self.down_right is not None):
            return self.down_right.height - self.down_left.height
        elif (self.down_left is None and self.down_right is not None):
            return self.down_right.height
        elif (self.down_left is not None and self.down_right is None):
            return -1 * self.down_left.height
        else:
            return 0

    def insert(self, value: int):
        if (self.value is None):
            self.value = value
        else:
            if (value < self.value):
                if (self.down_left is None):
                    self.down_left = node(value, up_right = self)
                else:
                    self.down_left.insert(value)
                self.updateheight()
            elif (value > self.value):
                if (self.down_right is None):
                    self.down_right = node(value, up_left = self)
                else:
                    self.down_right.insert(value)
                self.updateheight()

    def remove(self, value: int):
        if (self.value is not None):
            if (value < self.value):
                if (self.down_left is not None):
                    self.down_left.remove(value)
                    self.updateheight()
            elif (value > self.value):
                if (self.down_right is not None):
                    self.down_right.remove(value)
                    self.updateheight()
            else:
                if (self.down_left is None and self.down_right is None):
                    if (self.up_left is None and self.up_right is None):
                        self.value = None
                    elif (self.up_left is None):
                        self.up_right.down_left = None
                    else:
                        self.up_left.down_right = None
                elif (self.down_right is not None):
                    if (self.down_right.down_left is None):
                        self.value = self.down_right.value
                        self.down_right = self.down_right.down_right
                    else:
                        cur_node = self.down_right
                        while (cur_node.down_left is not None):
                            cur_node = cur_node.down_left
                        self.value = cur_node.value
                        cur_node.up_right.down_left = cur_node.down_right
                        while (cur_node.up_right is not None):
                            cur_node = cur_node.up_right
                            cur_node.updateheight()
                else:
                    if (self.down_left.down_right is None):
                        self.value = self.down_left.value
                        self.down_left = self.down_left.down_left
                    else:
                        cur_node = self.down_left
                        while (cur_node.down_right is not None):
                            cur_node = cur_node.down_right
                        self.value = cur_node.value
                        cur_node.up_left.down_right = cur_node.down_left
                        while (cur_node.up_left is not None):
                            cur_node = cur_node.up_left
                            cur_node.updateheight()
                self.updateheight()

    def updateheight(self):
        if (self.down_left is not None and self.down_right is not None):
            self.height = max(self.down_right.height, self.down_left.height) + 1
        elif (self.down_left is None and self.down_right is not None):
            self.height = self.down_right.height + 1
        elif (self.down_left is not None and self.down_right is None):
            self.height = self.down_left.height + 1
        else:
            self.height = 1
        if (self.get_balance() == 2):
            if (self.down_right.get_balance() == 1
                or self.down_right.get_balance() == 0):
                self.rotate_left()
            elif (self.down_right.get_balance() == -1):
                self.down_right.rotate_right()
        elif (self.get_balance() == -2):
            if (self.down_left.get_balance() == -1
                or self.down_left.get_balance() == 0):
                self.rotate_right()
            elif (self.down_left.get_balance() == 1):
                self.down_left.rotate_left()

    def rotate_right(self):
        if (self.up_left is None and self.up_right is None):
            self.up_left = self.down_left
            self.down_left = self.up_left.down_right
            self.up_left.up_right = None
            self.up_left.down_right = self
            if (self.down_left is not None):
                self.down_left.up_right = self
                self.down_left.up_left = None
        elif (self.up_left is not None):
            self.up_left.down_right = self.down_left
            self.down_left.up_left = self.up_left
            self.up_left = self.down_left
            self.down_left = self.up_left.down_right
            self.up_left.up_right = None
            self.up_left.down_right = self
            if (self.down_left is not None):
                self.down_left.up_right = self
                self.down_left.up_left = None
        else:
            self.up_right.down_left = self.down_left
            self.down_left.up_right = self.up_right
            self.up_right = None
            self.up_left = self.down_left
            self.down_left = self.up_left.down_right
            self.up_left.down_right = self
            if (self.down_left is not None):
                self.down_left.up_right = self
                self.down_left.up_left = None
        cur_node = self
        while (cur_node is not None):
            cur_node.updateheight()
            if (cur_node.up_left is not None):
                cur_node = cur_node.up_left
            elif (cur_node.up_right is not None):
                cur_node = cur_node.up_right
            else:
                break

    def rotate_left(self):
        if (self.up_left is None and self.up_right is None):
            self.up_right = self.down_right
            self.down_right = self.up_right.down_left
            self.up_right.up_left = None
            self.up_right.down_left = self
            if (self.down_right is not None):
                self.down_right.up_left = self
                self.down_right.up_right = None
        elif (self.up_left is not None):
            self.up_left.down_right = self.down_right
            self.down_right.up_left = self.up_left
            self.up_left = None
            self.up_right = self.down_right
            self.down_right = self.up_right.down_left
            self.up_right.down_left = self
            if (self.down_right is not None):
                self.down_right.up_left = self
                self.down_right.up_right = None
        else:
            self.up_right.down_left = self.down_right
            self.down_right.up_right = self.up_right
            self.up_right = self.down_right
            self.down_right = self.up_right.down_left
            self.up_right.up_left = None
            self.up_right.down_left = self
            if (self.down_right is not None):
                self.down_right.up_left = self
                self.down_right.up_right = None
        cur_node = self
        while (cur_node is not None):
            cur_node.updateheight()
            if (cur_node.up_left is not None):
                cur_node = cur_node.up_left
            elif (cur_node.up_right is not None):
                cur_node = cur_node.up_right
            else:
                break

class avl:
    def __init__(self):
        self.data = node()

    def insert(self, value: int):
        self.data.insert(value)
        self.checktop()

    def remove(self, value: int):
        self.data.remove(value)
        self.checktop()

    def print_tree(self):
        self.data.print_tree()

    def checktop(self):
        if (self.data.up_left is not None):
            self.data = self.data.up_left
        elif (self.data.up_right is not None):
            self.data = self.data.up_right
```

10 References